CS Cache Engine: Data Access Accelerator for Location-Based Services in Mobile Environments

Ken C. K. Lee[†], Wang-Chien Lee[†], Julian Winter[†], Baihua Zheng[‡] and Jianliang Xu[§]

[†]Department of Computer Science and Engineering, Pennsylvania State University, PA16802, USA.
[‡]School of Information Systems, Singapore Management University, Singapore.
[§]Department of Computer Science, Hong Kong Baptist University, Hong Kong.

{cklee,wlee,jwinter}@cse.psu.edu, bhzheng@smu.edu.sg, xujl@comp.hkbu.edu.hk

ABSTRACT

Location-based services (LBSs) have emerged as one of the killer applications for mobile and pervasive computing. Due to limited wireless channel bandwidth and scarce client resources, client-side data caching is essential to enhance the data availability and to improve the data access time. In this demonstration, we present a CS Cache Engine for LBS. The engine adopts our *Complementary Space Caching* (CS caching) scheme [3] that differs from conventional data caching schemes by preserving a global view of the database in the cache. The global view consists of cached objects and Complementary Regions (CRs) representing those objects in the server but not in the cache. The CS Cache Engine supports various location-based queries. In addition, it allows the clients to determine whether queries are completely answerable by the cache, thereby effectively avoiding unnecessary traffic over the wireless channel. In this paper, the architecture and the functionality of the CS Caching Engine are discussed. Furthermore, a tourist information application called TravelGuide powered by this cache engine is demonstrated.

1. INTRODUCTION

Location-based services (LBSs) have emerged as one of the killer applications for mobile and pervasive computing. Efficient processing of various location-based queries (with respect to the positions of users) is particularly critical to the provision of LBSs. For instance, queries like "Where is the nearest gas station?" and "Which ATMs are within 1 mile from my current position?" are very often asked by users. The former one is called the Nearest Neighbor (NN) query and the latter one is called the Range query. They are common types of *location-dependent queries*.

Typically, mobile computing applications are restricted by low-quality communication, frequent network disconnections and the limited resources of mobile devices. To improve data access efficiency and to alleviate the contention of wireless bandwidth, some data are cached at the client side. Queries can be answered by clients locally if the required data are available.

Client-side cache for LBSs mandates two important requirements to maximize the cache utilization and reduce the wireless channel contention, including:

(1) *supporting multiple kinds of location-dependent queries*.

(2) *determining whether a query can be answered locally at the client in order to reduce requests to the server*.

Existing caching schemes maintain a portion of a database in terms of pages, data records or previous query results [2]. Cached pages/records alone cannot help determine if a query can be locally answered, while cached query results are mostly constrained to supporting queries of specific types only. Thus, there is a strong need for devising new caching schemes to meet the above mentioned requirements. *Complementary Space Caching* (CS caching) [3] is a novel caching scheme we have recently proposed. In this paper, we present a *CS Caching Engine* based on CS caching scheme. The cache engine caches spatial data in the client to support various kinds of location-dependent queries issued by client applications. It consists of a query processor, a cache manager and cache memory. The query processor evaluates location-dependent queries with cached data and other metadata provided by the cache manager. The cache manager downloads additional data as needed and manages the cached data and metadata in the cache memory.

CS caching differs from existing caching schemes by preserving a global view of the database. In addition to the cached data objects, it maintains Complementary Regions (CRs) representing auxiliary information about *missing objects*, i.e., the objects that are not cached but available in the server. Figure 1 depicts an example where CS caching is adopted (in this figure, all dots, either in black or white, constitute the whole dataset, with black ones representing cached objects, and rectangular boxes enclosing white dots standing for the CRs in the cache). Suppose that three objects e, f and g (the result of the first query, Q_1) are cached and other objects (i.e., a, b, c, d, h and i) are abstracted as CRs in the cache. A range query Q_2, covering only object f and no CRs, can be locally asserted to require no additional object from the server. Therefore, no request to the server is needed and the query is completely answerable by the cache. Similarly, a nearest neighbor (NN) query Q_3, issued at p and with 'e' as the result, can also be answered locally without contacting the server. This caching scheme supports both Q_2 and Q_3, two queries of different types. These features are not available in existing caching schemes.
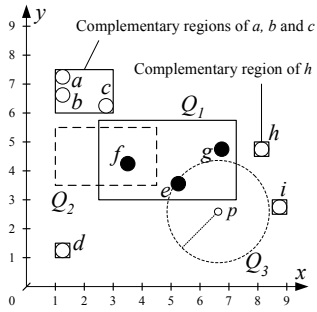
Figure 1: Complementary Space Caching

We prototype a tourist information application, called TouristGuide, for Manhattan, New York City to showcase the main functionalities of our CS Cache Engine, that is 1) supporting mixed types of queries, and 2) determining additional data objects as needed by queries. The remainder of this paper is organized as follows. Section 2 briefly reviews the CS caching model and the functionality of the CS Cache Engine. Section 3 describes the prototype presentation and demonstration of TravelGuide.

2. COMPLEMENTARY SPACE CACHING

2.1 Overview

CS caching maintains a global view of a database. As cache memory space is limited, different portions of the database are maintained in the cache in different granularity, which is determined by the access probabilities of corresponding data objects. The data objects with very high access probabilities are cached in the finest granularity, i.e., the *physical data objects* themselves. On the other hand, data objects less likely to be accessed in the near future are not cached, but are abstracted and logically cached in the form of *Complementary Regions* (CRs) in a coarse granularity. CRs present auxiliary information about *missing objects*, i.e., those objects not cached. The auxiliary information can facilitate local processing of various location-dependent queries and alleviate unnecessary queries to the server. CRs can be determined efficiently by the server based on minimum bounding boxes (MBBs) of R-tree index [1].
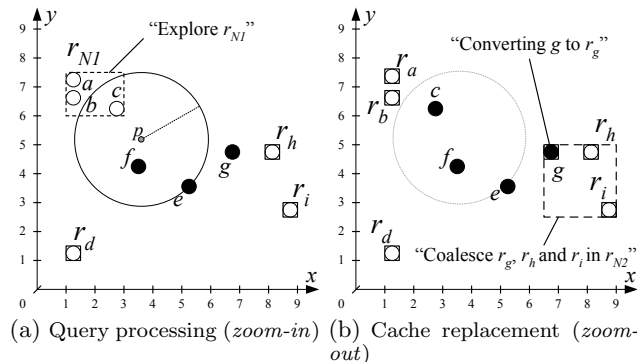

(a) Query processing (*zoom-in*) (b) Cache replacement (*zoom-out*)

Figure 2: CSC operations

Query processing and cache management in CS caching are different from those in existing caching schemes. Figure 2 (using the same settings as in Figure 1) outlines both the query processing and the cache replacement mechanisms. Suppose the cache of a client contains three objects, $\{e, f, g\}$ and four CRs, $\{r_{N1}, r_h, r_i, r_d\}$. As shown in Figure 2(a),

when the user moves to a position p and issues a query that partially covers r_{N1}, it implies the presence of some answer objects in the server. Consequently, the client explores r_{N1} for missing objects by querying the server. Then, object c, together with CRs r_a and r_b, is received. r_{N1} is replaced with c, r_a and r_b in the cache and hence the portion covered by r_{N1} is presented in a finer granularity. In that sense, query processing resembles a *zoom-in* operation that brings more details about the queried area into the cache. Finally, objects c, f and e are returned as the answers.

On the other hand, new objects and/or finer CRs may require the cache to free memory space for accommodation. Suppose an object g is chosen to be removed from the cache as shown in Figure 2(b). In order to preserve the global view, CR r_g is created to replace g in the same position and then g is physically deleted. Further, if more free space is needed, three closely located CRs, r_g, r_h and r_i, are coalesced into a single CR of a coarser granularity. This cache replacement is analogous to a *zoom-out* operation that removes the details of an area that the client is not interested in. At last, the cache contains three objects, $\{c, e, f\}$, and four CRs, $\{r_a, r_b, r_{N2}, r_d\}$.

2.2 Query Processing

Queries are processed locally by the query processor of the CS Cache Engine. It takes two steps, namely (1) *cache probing* and (2) *remainder query processing*. Cache probing examines the cache and figure out which CRs are needed to explore when a query is issued. Remainder query processing is invoked to submit a remainder query to fetch additional objects and refined CRs from the server when there are CRs needed to explore.

2.3 Cache Management

Cache management is critical to the cache performance in terms of access latency and cache hits. We focus on two aspects of cache management in this work, namely, *cache organization* that structures the cache content, and *cache replacement* that determines which object or CRs to purge from the cache in order make room for incoming objects and CRs.

Cache memory organization. The cache memory is organized as a cache table with each table entry storing one cached object (called *object entry*) or up to 'n' CRs (called *CR entry*). The object admission is performed by accommodating an object in a vacant entry. A CR entry contains one MBB which encloses all the CRs stored in the entry, and the admission of CRs is described as follows. In order to admit newly inserted CRs, the MBB associated with a CR entry has to be enlarged in order to bound new CRs. Among all the CR entries, the one with the least MBB enlargement is chosen to store the admitted CRs. This admission mechanism intends to group closely located CRs together in an entry, thereby providing fast CR lookup and facilitating CR coalescence to be discussed next. If a CR entry overflows (i.e., the number of CRs exceeds n) after insertion, all CRs are migrated to other CR entries with free slots. If no enough free slots are available to accommodate the CRs, a new entry is allocated and half of the CRs in the overflowed entry are moved to the new entry. Deletion removes a CR from an entry. To retain high occupancy, CRs in an under-full CR entry (i.e., its occupancy below $n/2$) are moved into other CR entries to release the under-full CR entry.

The CS Cache Engine provides two space allocation schemes, namely *static allocation* and *dynamic allocation*. For *static allocation*, the cache table is split into two portions with one dedicated to cached objects and the other to CRs, and the management of two portions is kept independent of each other. *Dynamic allocation* does not allocate objects and CRs in separate storage and treats them in the same way in order to exploit higher flexibility in space utilization. To improve the lookup efficiency, R-tree indexes are used. For state allocation, two R-tree indexes are built; one is built upon object entries and the other is built upon CR entries. For dynamic allocation, only one R-tree is built for the entire cache table.

Cache CR coalescence. Cache CR coalescence (i.e., replacing multiple CRs with one single CR) occurs when some CRs are less likely to be accessed and space is reclaimed for newly admitted CRs or objects. The efficiency of CR coalescence is crucial to the client performance when cache replacement occurs frequently. To meet this requirement, we adopt a pre-clustering technique to group CRs in the same CR entry at the admission time of a CR, instead of computing a coalesced CR based on a number of possible combinations at run time which may cause a serious performance bottleneck. When new space is needed, one CR entry is selected. A coalesced CR, based on the MBB associated with the selected CR entry, is formed and inserted to another CR entry, followed by removal of the selected CR entry.

Cache replacement mechanism. Cache replacement in CS caching is not only for fitting objects and CRs in the cache but also for balancing the granularity of the global view (in terms of objects and CRs) maintained in cache. An object removal is performed as converting the object to a CR. A CR removal implies coalescence with other CRs. In the following, we discuss the replacement mechanisms for both static allocation and dynamic allocation.

- **Static allocation.** Replacement starts in the object portion. If the object portion is full, victim objects are removed by transforming them into CRs, which are then inserted to the CR portion. If a CR portion is full, victim CR entries are chosen to coalesce. The victim selection (i.e. cache replacement policy) is based on FAR [4] heuristics in our implementation. For the FAR heuristic, distance is measured between the current client position and the anticipated CR (either formed from object deletion or coalescence of CRs).

- **Dynamic allocation.** Both object replacement and CR coalescence are considered to make room for caching new objects and/or CRs. In order to prioritize object replacement and CR coalescence which are totally different in nature, a unified replacement score based on the download cost is used [3]. A cluster of CRs with the lowest score is coalesced. An object with the lowest score is converted to a CR. The newly formed CR is inserted back into the cache.

3. DEMO DESCRIPTION

3.1 Prototype Implementation

Our CS Cache Engine is currently implemented as a Dynamic Link Library (DLL) and runs on a PocketPC operated by Windows Mobile 5.0. The CS Cache Engine is dynamically loaded upon the launch of the supported location-based application(s). Our prototype platform is equipped with GPS and a WiFi network interface card. From GPS, the CS Cache Engine obtains current locations and keeps track of the user movement. In addition to the built-in PocketPC memory, the CS caching engine stores data in external memory. The client and server communicate through WiFi. Our CS Cache Engine is configurable using the system registry, e.g., cache storage size and storage strategy. On the server side, we developed our Spatial DB using Berkeley DB as our server database running on Windows platform. We implement the server operations to support both query processing and CR manipulation.

3.2 Demonstration

The functionality of the CS Cache Engine is demonstrated through a tourist guide prototype called TravelGuide. This application aims at providing location-dependent information to a tourist. With the data access part supported by the CS Cache Engine, TravelGuide functions as an interface to accept user queries and to display results. With GPS, the position of a tourist is tracked and provided for queries[1]. The example run allows a tourist in Manhattan NYC to specify attractive points of interest, e.g. a cafe or restaurant, the nature of search (either nearest one or those within a specified distance) and associated parameters. Such parameters are taken and expressed as a query to the underlying CS Cache Engine.

Acknoledgements

4. REFERENCES

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25*, pages 322–331, 1990.

[2] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *Proceedings of 22th International Conference on Very Large Data Bases (VLDB), Mumbai (Bombay), India, Sep 3-6*, pages 330–341, 1996.

[3] K. C. Lee, W.-C. Lee, B. Zheng, and J. Xu. Caching Complementary Space for Location-Based Services. In *Proceedings of the 10th International Conference on Extending Database Technology (EDBT), Munich, Germany, Mar 26-31*, pages 1020–1038, 2006.

[4] Q. Ren and M. H. Dunham. Using Semantic Caching to Manage Location Dependent Data in Mobile Computing. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking, Boston, MA, USA, Aug 6-11*, pages 210–221, 2000.

---

[1]In the time of demonstration, we use the GPS emulator that captures stylus movement on screen as user movement on the ground.